SMART CONTRACT AUDIT REPORT

for

Carbon Protocol

Prepared By: Xiaomi Huang

PeckShield

April 4, 2023

Document Properties

Client	Bancor
Title	Smart Contract Audit Report
Target	Carbon Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 4, 2023	Xuxian Jiang	Final Release
1.0-rc	April 3, 2023	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Carbon protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Carbon

The Carbon protocol is a next-generation AMM which introduces the key new feature of asymmetric liquidity, wherein any given bonding curve only trades in a single direction, effectively being an out-of the-money limit order. Carbon offers unique trading abilities which enable adjustable trading strategies with custom ranges, linked orders, and continuous one-directional liquidity. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Carbon Protocol

Item	Description
Issuer	Bancor
Website	https://www.carbondefi.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 4, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/bancorprotocol/carbon-contracts.git (e9d8b4a)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/bancorprotocol/carbon-contracts.git (6ddeace)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Carbon` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	2	■■
Informational	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Validation of Actions to Calculate Trade Amounts	Business Logic	Fixed
PVE-002	Low	Revisited Refund of Native Token in Strategies	Business Logic	Fixed
PVE-003	Medium	Trust on Admin Keys	Security Features	Mitigated
PVE-004	Low	Suggested Liquidity Validation for Gas Efficiency	Coding Practices	Confirmed
PVE-005	Informatinoal	Suggested Use of InsufficientLiquidity() Error	Coding Practices	Fixed

Beside the identified issues, we note that the staking support assumes the staked tokens are not deflationary. Also, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Validation of Actions to Calculate Trade Amounts

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Strategies`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Carbon` protocol provides a pair of interfaces for traders to query the total source and target amounts for a trade. The trade can be fulfilled by the given trade actions in the same pool, and the query is completed by simulating the trade in each of the target orders from the trade actions. While examining the implementation of the query, we notice there is a lack of validation for the given trade actions which may be from different pools.

To elaborate, we show below the affected `_tradeSourceAndTargetAmounts()` routine. It loops all the given trade actions (`tradeActions`) and calculates the source and target amounts for the single trade action by calling the `_singleTradeActionSourceAndTargetAmounts()` routine (line 549). Per design, the given trade actions shall be from the same pool as the desired source and target tokens. However, we notice there is a lack of validation for the given trade actions. As a result, if some trade action is from a different pool, the calculated source and target amounts are wrong. Based on this, we suggest to add a validation for the trade actions, and ensure they are from the same pool as the trade.

```
535     function _tradeSourceAndTargetAmounts(
536         TradeTokens memory tokens,
537         TradeAction[] calldata tradeActions,
538         Pool memory pool,
539         bool byTargetAmount
540     ) internal view returns (SourceAndTargetAmounts memory totals) {
541         // process trade actions
542         for (uint256 i = 0; i < tradeActions.length; i = uncheckedInc(i)) {
```

```
            // prepare variables
            uint256[3] memory packedOrdersMemory = _packedOrdersByStrategyId[
                tradeActions[i].strategyId];
            (Order[2] memory orders, bool ordersInverted) = _unpackOrders(
                packedOrdersMemory);

            // calculate the orders new values
            uint256 targetTokenIndex = _findTargetOrderIndex(pool, tokens,
                ordersInverted);
            SourceAndTargetAmounts memory tempTradeAmounts =
                _singleTradeActionSourceAndTargetAmounts(
                orders[targetTokenIndex],
                tradeActions[i].amount,
                byTargetAmount
            );

            // update totals
            totals.sourceAmount += tempTradeAmounts.sourceAmount;
            totals.targetAmount += tempTradeAmounts.targetAmount;
        }

        // apply trading fee
        if (byTargetAmount) {
            totals.sourceAmount = _addFee(totals.sourceAmount);
        } else {
            totals.targetAmount = _subtractFee(totals.targetAmount);
        }
    }
```

Listing 3.1: `Strategies::_tradeSourceAndTargetAmounts()`

Recommendation Add a proper validation for the given trade actions to ensure they are from the same pool as the trade.

Status This issue has been fixed in this commit: `4eca565`.

3.2 Revisited Refund of Native Token in Strategies

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Strategies
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Carbon protocol provides a general interface to pull users deposits into the protocol. Users deposits can be of normal ERC20 token or the native token. In case of the native token, if the transferred-in token amount (msg.value) is larger than the deposit amount, the remaining is refunded to the user. While reviewing the scenarios of refund, we notice it does not consider the possible refund when the deposit amount is 0.

To elaborate, we show below the code snippet of the _validateDepositAndRefundExcessNativeToken() routine. As the name indicates, it validates users deposits and refunds the excess native token. Currently the refund only happens when the deposit amount is not 0. In particular, if the deposit amount is 0 (line 633), it directly returns from the function. In this case, if the user takes in some amount of the native token (msg.value > 0), they are locked in the protocol. Based on this, we suggest to check and refund the possible native token to the user as well even when the deposit amount is 0 .

```
627     function _validateDepositAndRefundExcessNativeToken(
628         Token token,
629         address owner,
630         uint256 depositAmount,
631         uint256 txValue
632     ) private {
633         if (depositAmount == 0) {
634             return;
635         }
636 
637         if (token.isNative()) {
638             if (txValue < depositAmount) {
639                 revert NativeAmountMismatch();
640             }
641 
642             // refund the owner for the remaining native token amount
643             if (txValue > depositAmount) {
644                 payable(address(owner)).sendValue(txValue - depositAmount);
645             }
646         } else {
647             token.safeTransferFrom(owner, address(this), depositAmount);
648         }
```

```
    }
```

Listing 3.2: Strategies :: _validateDepositAndRefundExcessNativeToken()

Recommendation Check and refund the possible native token when the deposit amount is 0.

Status This issue has been fixed in this commit: `9a94d14`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Carbon` protocol, there is a special `admin` account (with the `ROLE_ADMIN`). This `admin` account plays a critical role in governing and regulating the protocol-wide operations (e.g., assign other roles, configure various settings). Our analysis shows that the `admin` account and other privileged roles need to be scrutinized. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

```
    function setTradingFeePPM(uint32 newTradingFeePPM) external onlyAdmin validFee(
        newTradingFeePPM) {
        _setTradingFeePPM(newTradingFeePPM);
    }

    function pause() external onlyRoleMember(ROLE_EMERGENCY_STOPPER) {
        _pause();
    }

    function unpause() external onlyRoleMember(ROLE_EMERGENCY_STOPPER) {
        _unpause();
    }

    function withdrawFees(
        uint256 amount,
        Token token,
        address recipient
    )
        external
        whenNotPaused
        onlyRoleMember(ROLE_FEES_MANAGER)
        validAddress(recipient)
        validAddress(Token.unwrap(token))
```

```
        greaterThanZero(amount)
        nonReentrant
    {
        _withdrawFees(msg.sender, amount, token, recipient);
    }
```

Listing 3.3: Example Privileged Operations in `CarbonController`

```
    function mint(address owner, uint256 tokenId) external onlyRoleMember(ROLE_MINTER) {
        _safeMint(owner, tokenId);
    }

    function burn(uint256 tokenId) external onlyRoleMember(ROLE_MINTER) {
        _burn(tokenId);
    }

    function setBaseURI(string memory newBaseURI) public onlyAdmin {
        __baseURI = newBaseURI;

        emit BaseURIUpdated(newBaseURI);
    }

    function setBaseExtension(string memory newBaseExtension) public onlyAdmin {
        _baseExtension = newBaseExtension;

        emit BaseExtensionUpdated(newBaseExtension);
    }

    function useGlobalURI(bool newUseGlobalURI) public onlyAdmin {
        if (_useGlobalURI == newUseGlobalURI) {
            return;
        }

        _useGlobalURI = newUseGlobalURI;
        emit UseGlobalURIUpdated(newUseGlobalURI);
    }
```

Listing 3.4: Example Privileged Operations in `Voucher`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged accounts to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirms they will use a multi-sig account as the admin.

3.4 Suggested Liquidity Validation for Gas Efficiency

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Strategies`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `Carbon` protocol provides an off-chain matching algorithm whereby a trade of a given number of tokens is spread over one or more orders to achieve the most favorable rate of exchange for the trader. The trade is fulfilled by performing a single trade action in each of the matched orders. While examining the gas cost to fulfill the trade, we notice the gas cost could be improved by validating the order liquidity before calculating the source and target amounts of a single trade action.

To elaborate, we show below the code snippet of the `_trade()` routine, which is used to fulfill the trade. It loops all the given trade actions, finds the target order in each trade action and calculates the source and target amounts for the single trade action. The source and target amounts of each single trade action is calculated by calling the `_singleTradeActionSourceAndTargetAmounts()` routine (line 419). The token balances in the source and target orders are updated accordingly per the calculated source and target amounts (line 426 – 427). In particular, if the liquidity is not sufficient in the target order, the update of `targetOrder.y` reverts due to underflow.

However, we notice the `_singleTradeActionSourceAndTargetAmounts()` routine is gas intensive, and the gas can be saved by stopping the trade ahead when the liquidity is not sufficient. Our analysis shows that, if the trade is based on target token, the target amount of the single action and the liquidity of the target order are both available to check if the liquidity is sufficient or not. If the liquidity is not sufficient, we suggest to stop the trade before calling the `_singleTradeActionSourceAndTargetAmounts()` routine.

```
391     function _trade(
392         TradeAction[] calldata tradeActions,
393         TradeParams memory params
394     ) internal returns (SourceAndTargetAmounts memory totals) {
395         // process trade actions
396         for (uint256 i = 0; i < tradeActions.length; i = uncheckedInc(i)) {
397             // prepare variables
398             uint256 strategyId = tradeActions[i].strategyId;
399             uint256[3] storage packedOrders = _packedOrdersByStrategyId[strategyId];
400             uint256[3] memory packedOrdersMemory = _packedOrdersByStrategyId[strategyId
                    ];
```

```
            (Order[2] memory orders, bool ordersInverted) = _unpackOrders(
                packedOrdersMemory);

            // make sure strategyIds match the provided source/target tokens
            if (_poolIdByStrategyId(strategyId) != params.pool.id) {
                revert InvalidTradeActionStrategyId();
            }

            // make sure all tradeActions are provided with a positive amount
            if (tradeActions[i].amount == 0) {
                revert InvalidTradeActionAmount();
            }

            // calculate the orders new values
            uint256 targetTokenIndex = _findTargetOrderIndex(params.pool, params.tokens,
                 ordersInverted);

            Order memory targetOrder = orders[targetTokenIndex];
            Order memory sourceOrder = orders[1 - targetTokenIndex];

            SourceAndTargetAmounts memory tempTradeAmounts =
                _singleTradeActionSourceAndTargetAmounts(
                targetOrder,
                tradeActions[i].amount,
                params.byTargetAmount
            );

            // update the orders with the new values
            targetOrder.y -= tempTradeAmounts.targetAmount;
            sourceOrder.y += tempTradeAmounts.sourceAmount;
            if (sourceOrder.z < sourceOrder.y) {
                sourceOrder.z = sourceOrder.y;
            }
            ...
        }
        ...
        return totals;
    }
```

Listing 3.5: Strategies::_trade()

Recommendation Revisit the above `_trade()` routine to stop the trade before calling the `_singleTradeActionSourceAndTargetAmounts()` routine when the trade is based on the target amount and the liquidity in the target order is not sufficient.

Status This issue has been confirmed by the team and they decides to leave it as is.

3.5 Suggested Use of InsufficientLiquidity() Error

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Strategies`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `Carbon` protocol, it defines an `InsufficientLiquidity()` error in the `Strategies` contract, but the error is not used any where. Our analysis shows the error could be reported when the liquidity of the target order is not sufficient for a trade.

In the following, we show the code snippet of the `_trade()` routine. It calculates the source and target amounts for each of the trade actions, and updates the source and target orders accordingly. In particular, if the liquidity in the target order is not sufficient, the update of the target order reverts because of the arithmetic underflow (line 426). However, the trade reverts without providing a proper error message to the trader. Based on this, we suggest to check if the liquidity of the target order is sufficient or not for the target amount, and revert the trade with the `InsufficientLiquidity()` error when the liquidity is not sufficient.

```
391     function _trade(
392         TradeAction[] calldata tradeActions,
393         TradeParams memory params
394     ) internal returns (SourceAndTargetAmounts memory totals) {
395         // process trade actions
396         for (uint256 i = 0; i < tradeActions.length; i = uncheckedInc(i)) {
397             // prepare variables
398             uint256 strategyId = tradeActions[i].strategyId;
399             uint256[3] storage packedOrders = _packedOrdersByStrategyId[strategyId];
400             uint256[3] memory packedOrdersMemory = _packedOrdersByStrategyId[strategyId
                    ];
401             (Order[2] memory orders, bool ordersInverted) = _unpackOrders(
                    packedOrdersMemory);
402
403             // make sure strategyIds match the provided source/target tokens
404             if (_poolIdByStrategyId(strategyId) != params.pool.id) {
405                 revert InvalidTradeActionStrategyId();
406             }
407
408             // make sure all tradeActions are provided with a positive amount
409             if (tradeActions[i].amount == 0) {
410                 revert InvalidTradeActionAmount();
411             }
412
413             // calculate the orders new values
```

```
            uint256 targetTokenIndex = _findTargetOrderIndex(params.pool, params.tokens,
                 ordersInverted);

            Order memory targetOrder = orders[targetTokenIndex];
            Order memory sourceOrder = orders[1 - targetTokenIndex];

            SourceAndTargetAmounts memory tempTradeAmounts =
                _singleTradeActionSourceAndTargetAmounts(
                targetOrder,
                tradeActions[i].amount,
                params.byTargetAmount
            );

            // update the orders with the new values
            targetOrder.y -= tempTradeAmounts.targetAmount;
            sourceOrder.y += tempTradeAmounts.sourceAmount;
            if (sourceOrder.z < sourceOrder.y) {
                sourceOrder.z = sourceOrder.y;
            }
            ...
        }
        ...
        return totals;
    }
```

Listing 3.6: Strategies :: _trade()

Recommendation Properly report the `InsufficientLiquidity()` error when the liquidity of the target order is not sufficient for the target amount.

Status This issue has been fixed in this commit: `2bbdcd5`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Carbon` protocol, which is a next-generation `AMM` protocol introducing the key new feature of asymmetric liquidity, wherein any given bonding curve only trades in a single direction, effectively being an out-of the-money limit order. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

PeckShield

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.

PeckShield